Games. Adversial search

Games and Adversarial Search

Today we'll examine the problems that arise when we try to plan ahead in a world where other agents are planning against us.

Topics:

- Games
- Optimal Decisions: Minimax algorithm
- α-β Pruning

Games

In multiagent environments, the unpredictability of other agents can introduce contingencies into the agent's problem solving process.

These other agents may be cooperative or competitive.

In competitive multiagent environments the agents' goals are in conflict and this gives rise to adversarial search problems - often known as games.

AI Games

AI games are usually of a rather specialised kind:

- Game theorist description: deterministic, turn-taking, two-player, zero-sum games, with perfect information.
- AI theorist description: deterministic, fully observable environments in which there are two agents whose actions must alternate and in which the utility values at the end of the game are always equal and opposite.

AI Games

We will consider games with two players: MAX and MIN

MAX moves first, and then they take turns moving until the game is over.

At the end of the game, points are awarded to the winning player and penalties are given to the one who lost.

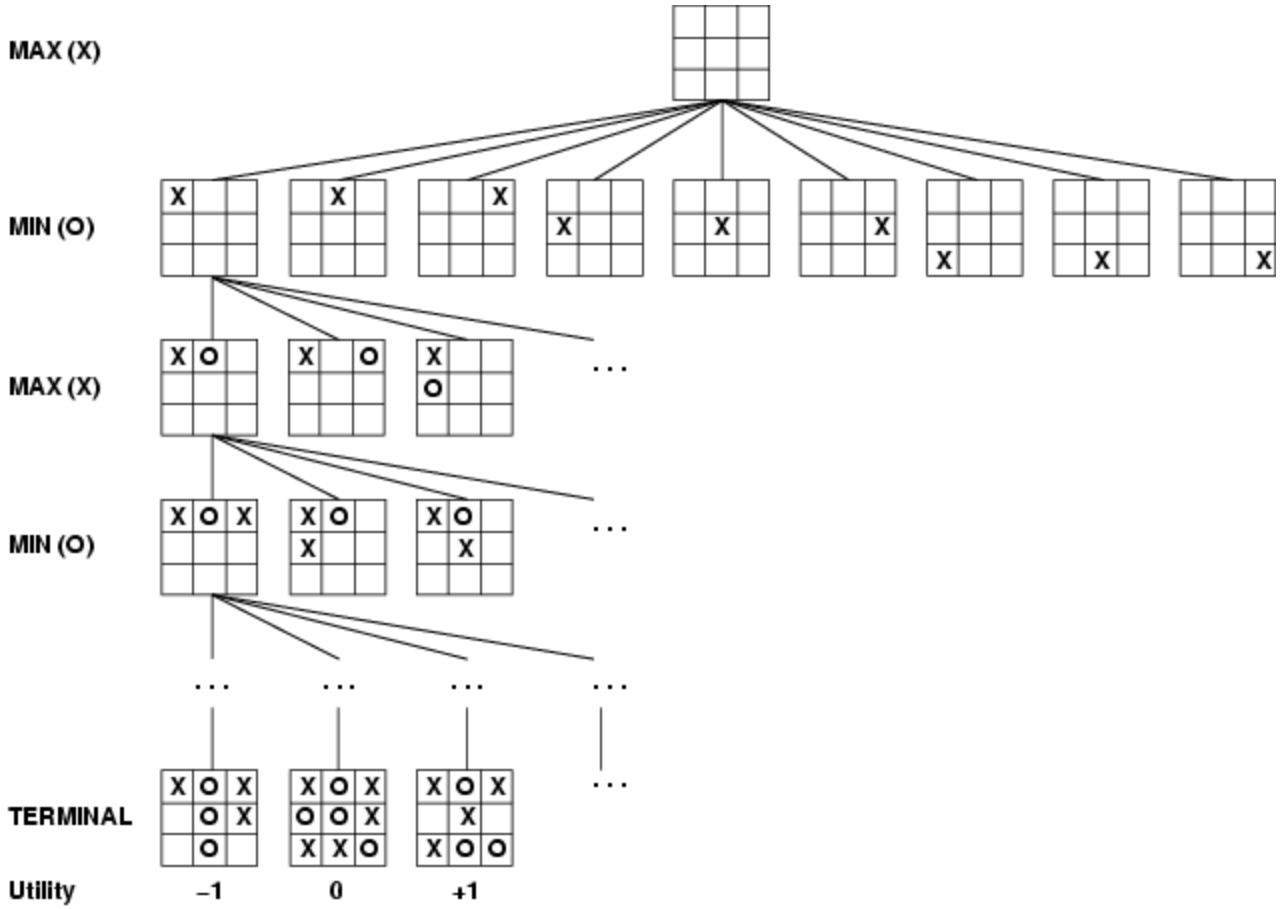
Games as Search

A game can be formally defined as a kind of search problem with the following components:

- The initial state, which includes the board position and identifies the player to move
- A successor function, which returns a list of (move, state) pairs, each indicating a legal move and the resulting state
- A terminal test, which determines when the game is over. States where the game has ended are called terminal states.
- A utility function which gives a numeric value for the terminal state.

The initial state and the legal moves for each side define the game tree.

Game tree (2-player, deterministic, turns)



Part of the game tree for tic-tac-toe. The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names).

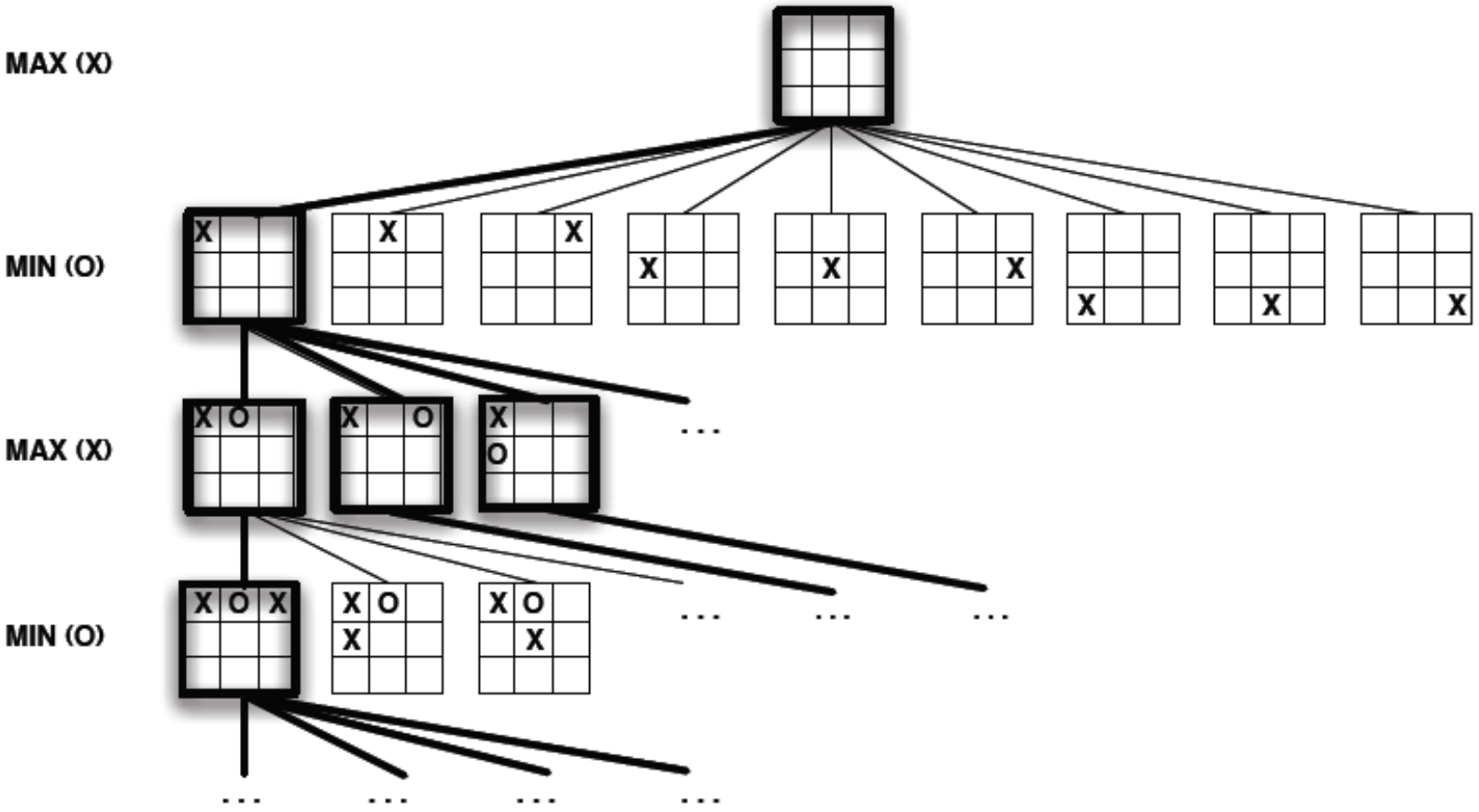
Optimal Strategies

In normal search problems the optimal solution is a sequence of moves leading to a goal state - a terminal state that is a win.

However, in a game MIN has something to say about what happens.

MAX therefore must find a contingent strategy, which specifies MAX's move for every possible game state resulting from MIN's move.

Optimal Strategy



Strategy: a *policy*, i.e., a function from states into actions that tells what you will do in every game state where it's your move.

In a 2-player deterministic turn-taking game, it's a subtree of the game tree

Optimal Strategies

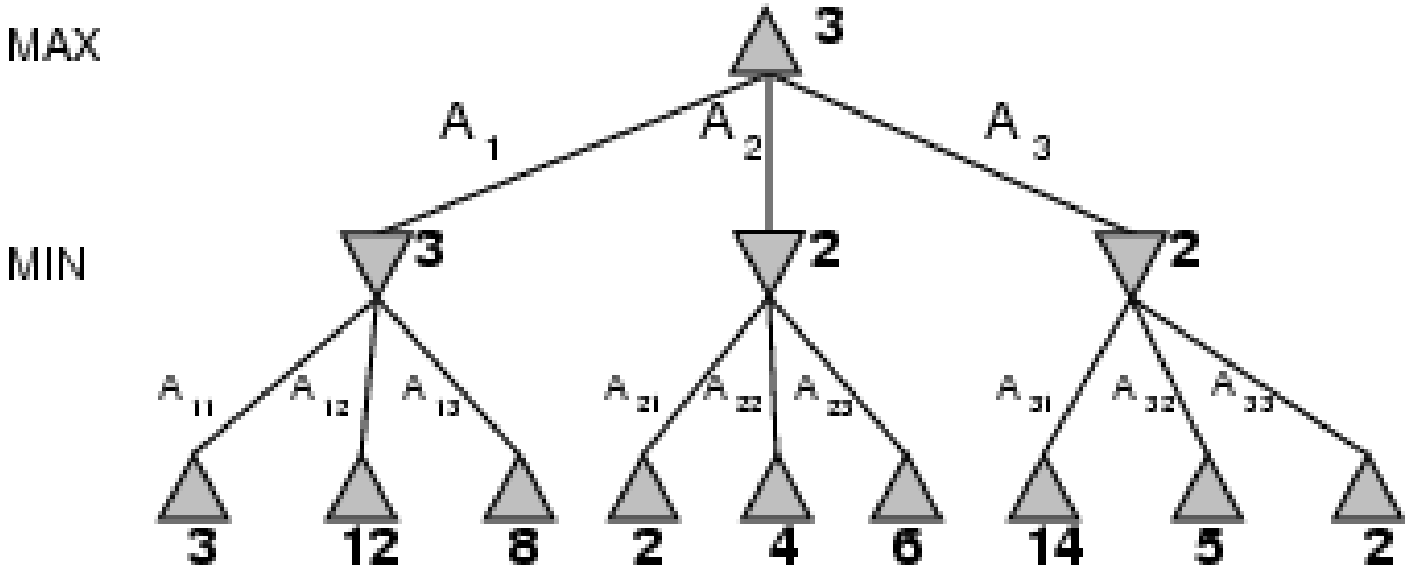
Roughly speaking, an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.

How do you compute an optimal strategy?

Optimal Strategies: Case Study

We will use the following trivial game to examine how to compute an optimal strategy:

- MAX has 3 possible initial moves A_1,A_2,A_3
- MIN's possible replies to A1 are A_{11}, A_{12}, A_{13}.
- The game ends after one move each by MAX and MIN (In game terms we say that this tree is one move deep, consisting of two half-moves, each of which is called a ply)
- The utilities of the terminal states in this game range from 2 to 14



Optimal Strategies: Minimax

Given a game tree, the optimal strategy can be determined by examining the minimax value of each node, MINIMAX-VALUE(*n*)

The minimax value of a node is the utility (for MAX) of being in the corresponding state, *assuming that both players play optimally from there to the end of the game*; i.e.: choose the state with the best achievable payoff against best player

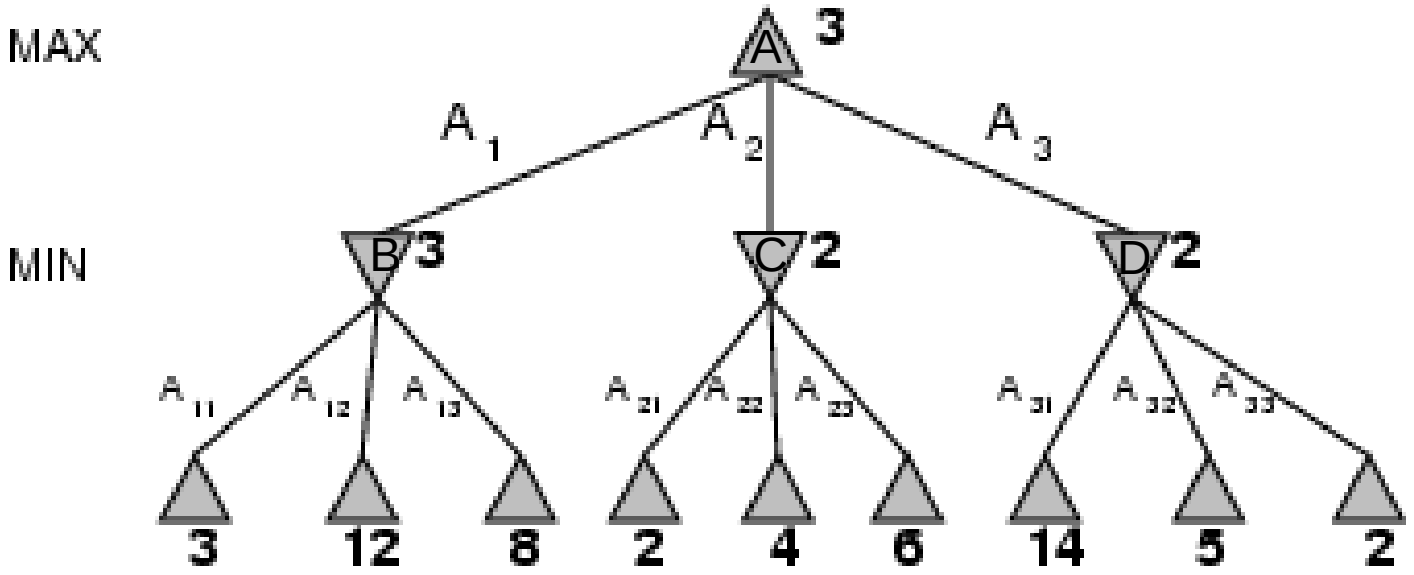
MINIMAX-VALUE(n) =

$$\begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\ \max_{s \in \text{Successors}(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in \text{Successors}(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MIN node} \end{cases}$$

Optimal Strategies: Minimax

Lets apply this approach to the case study game tree

- The terminal nodes are already labeled with their utility values.
- The first MIN node, B, has three successors, with values 3, 12, and 8, so its minimax value is 3, similarly C and D have minimax value 2
- The root node is a MAX node; its successors have minimax values 3, 2 and 2; so it has a minimax value of 3



Optimal Strategies: Minimax algorithm

The minimax algorithm computes the minimax decision from the current state.

It uses a simple recursive computation of the minimax value of each successor state, directly implementing the defining equations.

The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are backed up through the tree as the recursion unwinds.

Optimal Strategies: Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

Returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility.

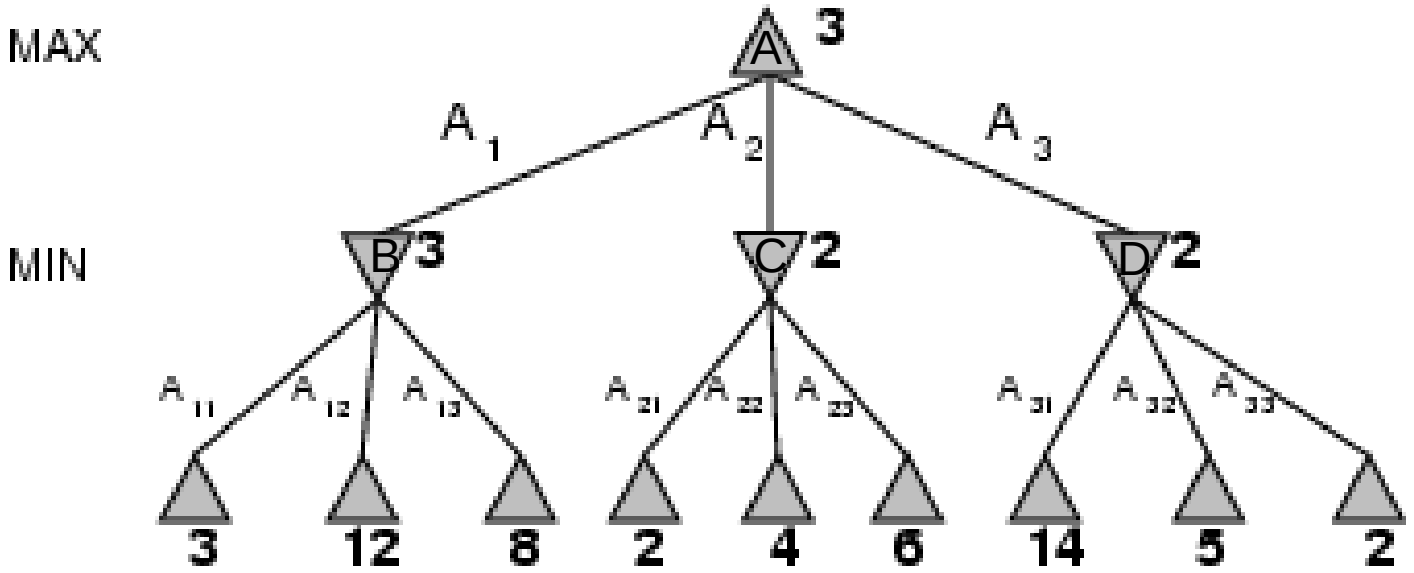
The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state.

Optimal Strategies: Minimax algorithm

In the example below, the minimax algorithm first recurses down to the three bottom-left nodes, and uses the UTILITY function on them to discover that their values are 3, 12, and 8 respectively.

Then it takes the minimum of these values, 3, and returns it as the back-up value of node B. A similar process give the backed up values of 2 for C and 2 for D.

Finally we take the max value of 3, 2, and 2 to get the backed up value of 3 for the root node.



Properties of minimax

Completeness

- Yes (if the tree is finite)

Optimal

- Yes (against optimal opponent)

Complexity: The minimax algorithm performs a complete depth-first exploration of the game tree. Let m be the maximum depth of the tree and b be the branching factor, then:

Space complexity

- Space complexity? O(bm)

Time complexity X

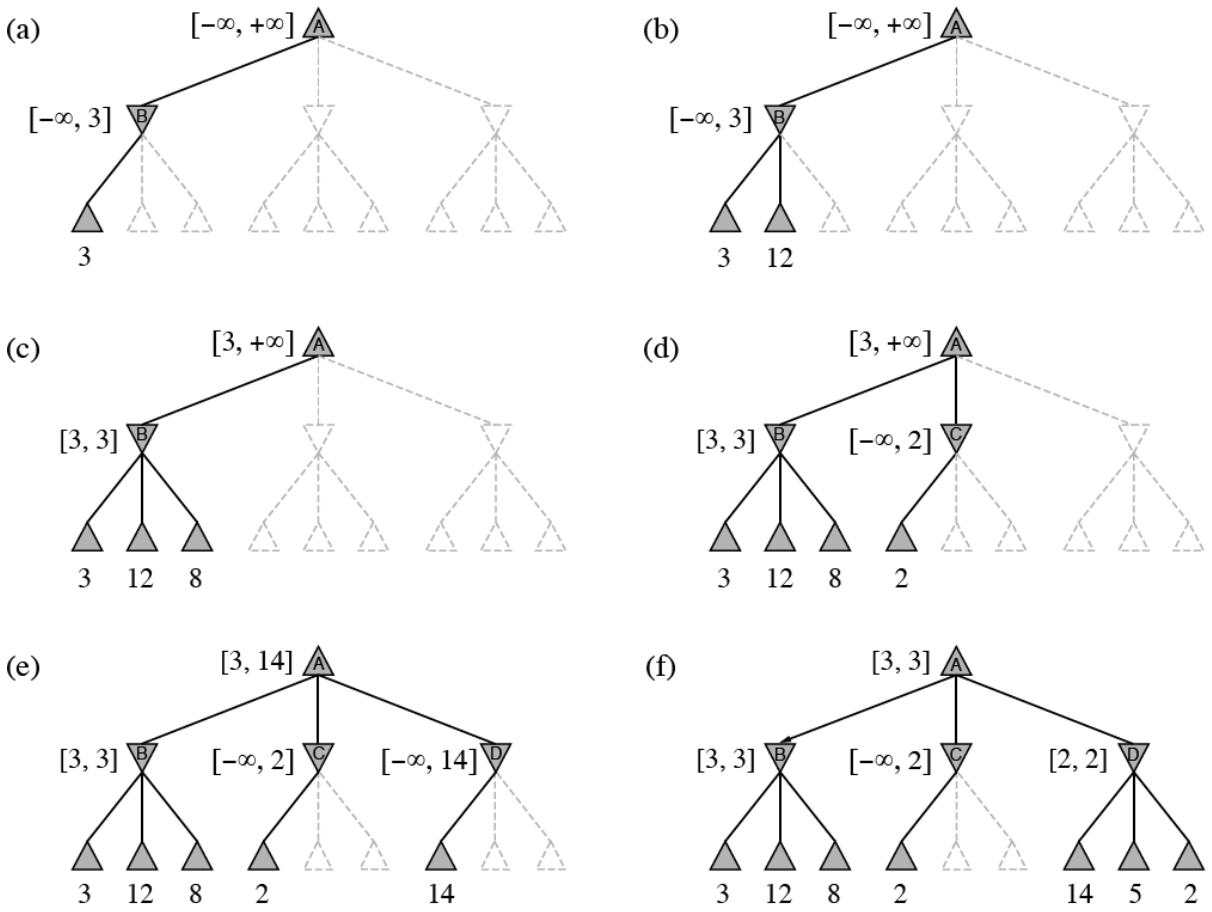
- $O(b^m)$ Recall that m can be much larger than d (the depth of the shallowest solution) and is infinite if the tree in unbounded.
- For real games, the time cost is totally impractical, e.g.: for a reasonable chess game, $b \approx 35$, $m \approx 100$

α-β pruning

α-β pruning when applied to a standard minimax tree returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

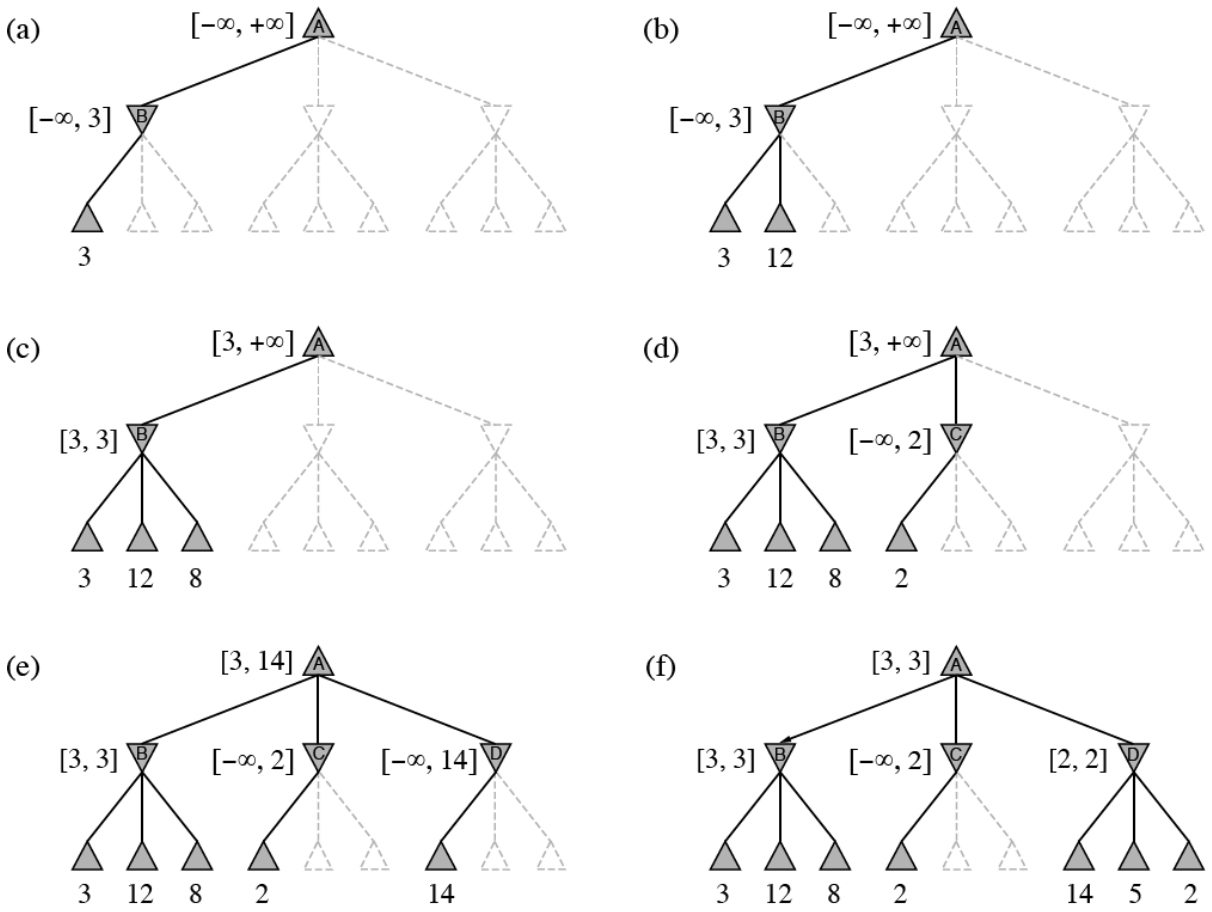
We will examine how alpha-beta pruning works by applying it to our example minimax game tree.

α-β pruning



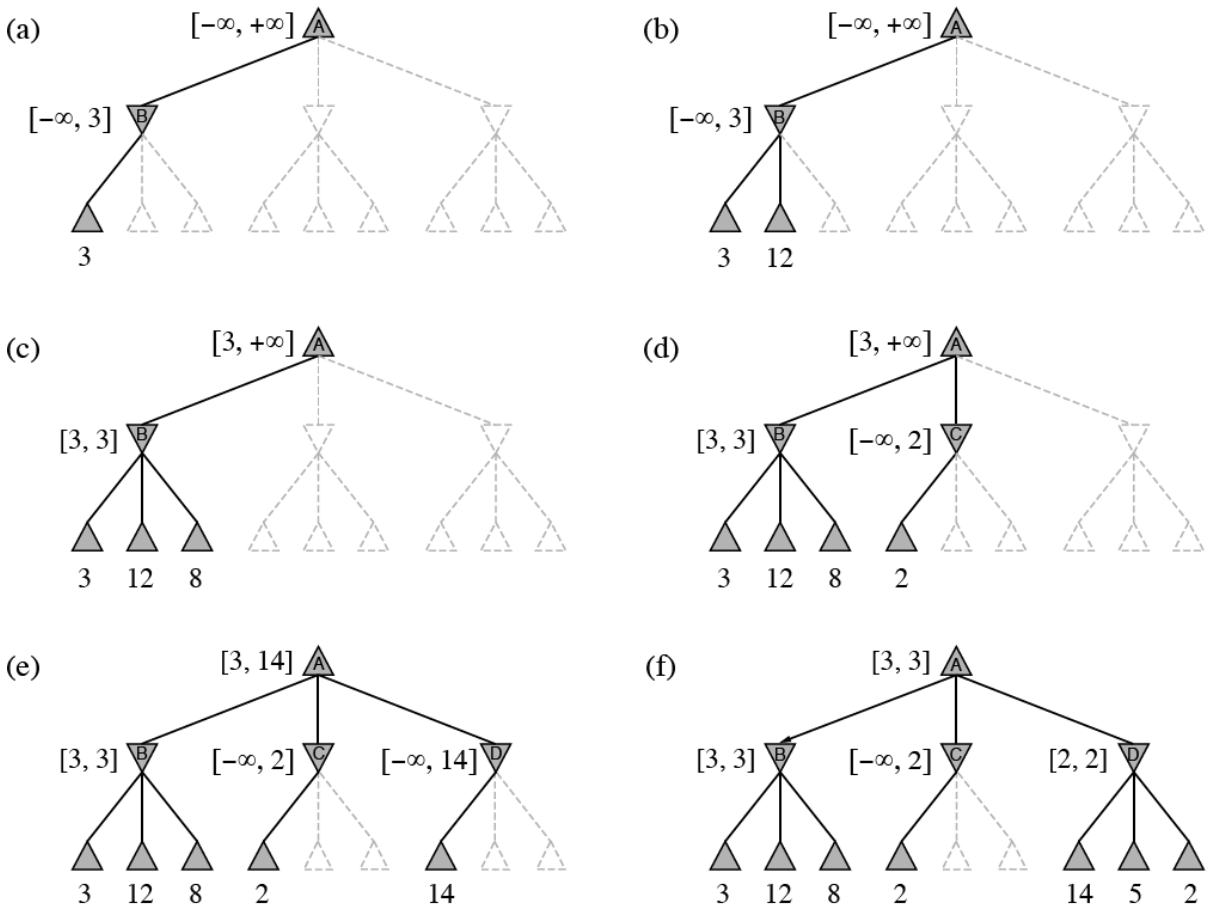
(a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3.

α-β pruning



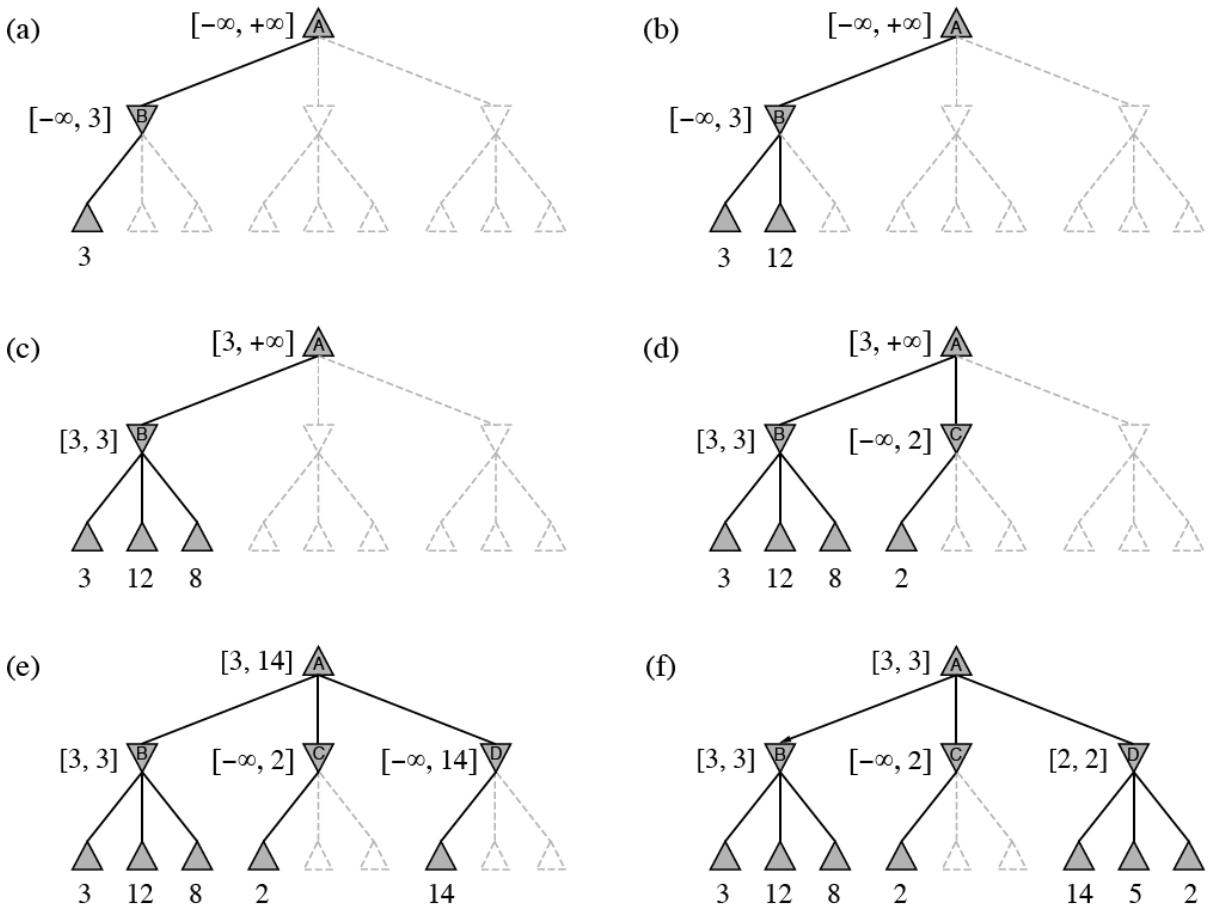
(b) The second leaf below B has a value of 12; MIN would avoid this move, so that value of B is still at most 3.

α-β pruning



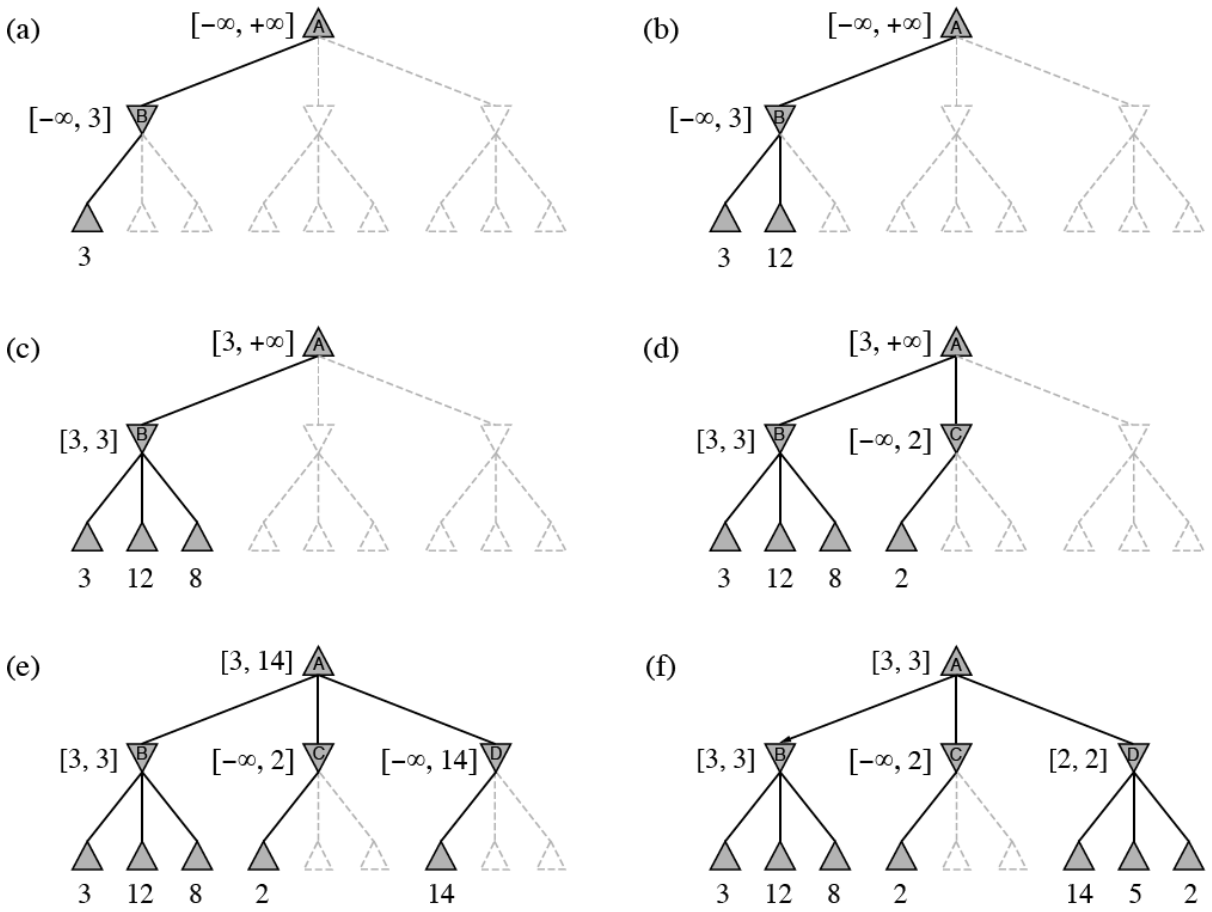
(c.) the third leaf below B has a value of 8; we have seen all B's successors, so the value of B is exactly 3. Now, we can infer that the value of the root is at least 3, because MAX has a choice worth 3 at the root.

α-β pruning



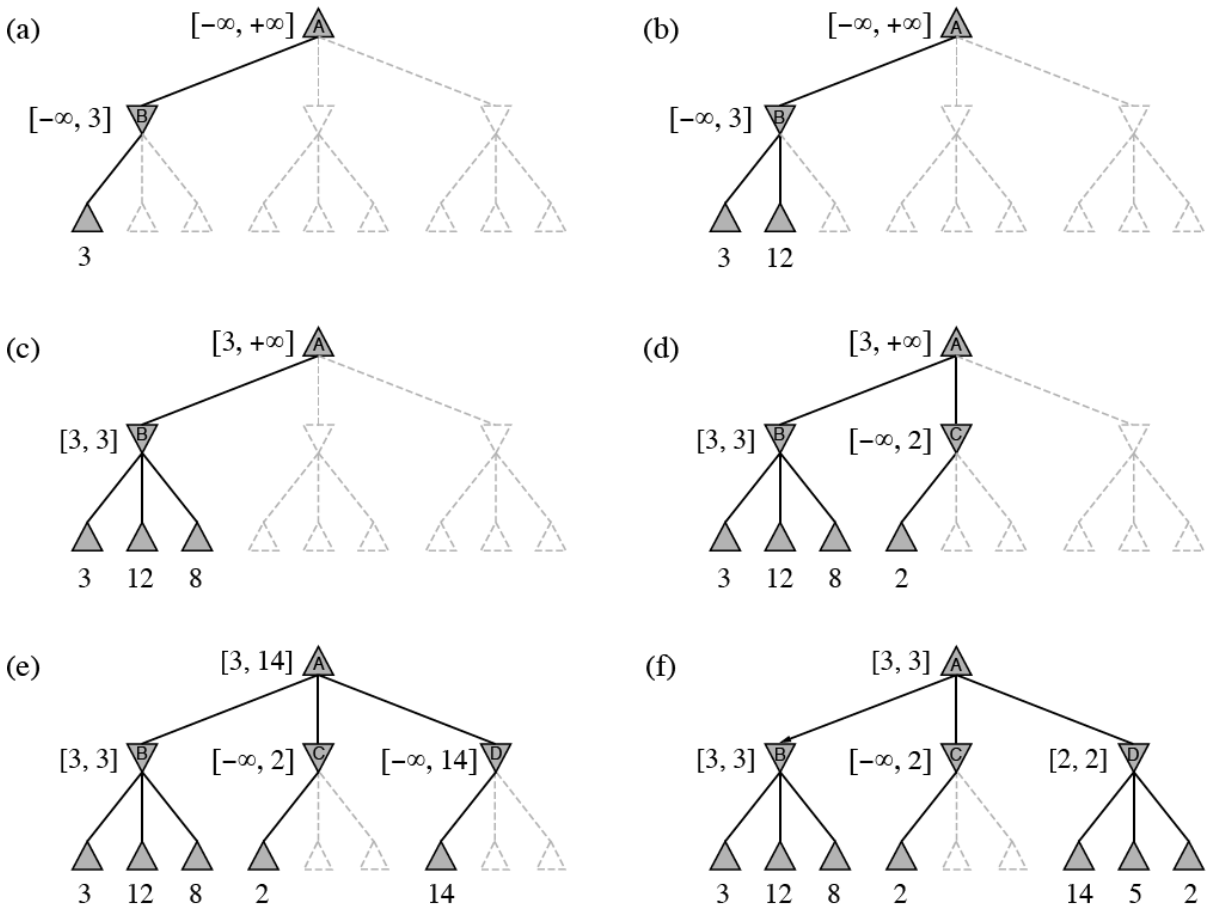
(d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of at most 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successors of C. This is an example of alpha-beta pruning.

α-β pruning



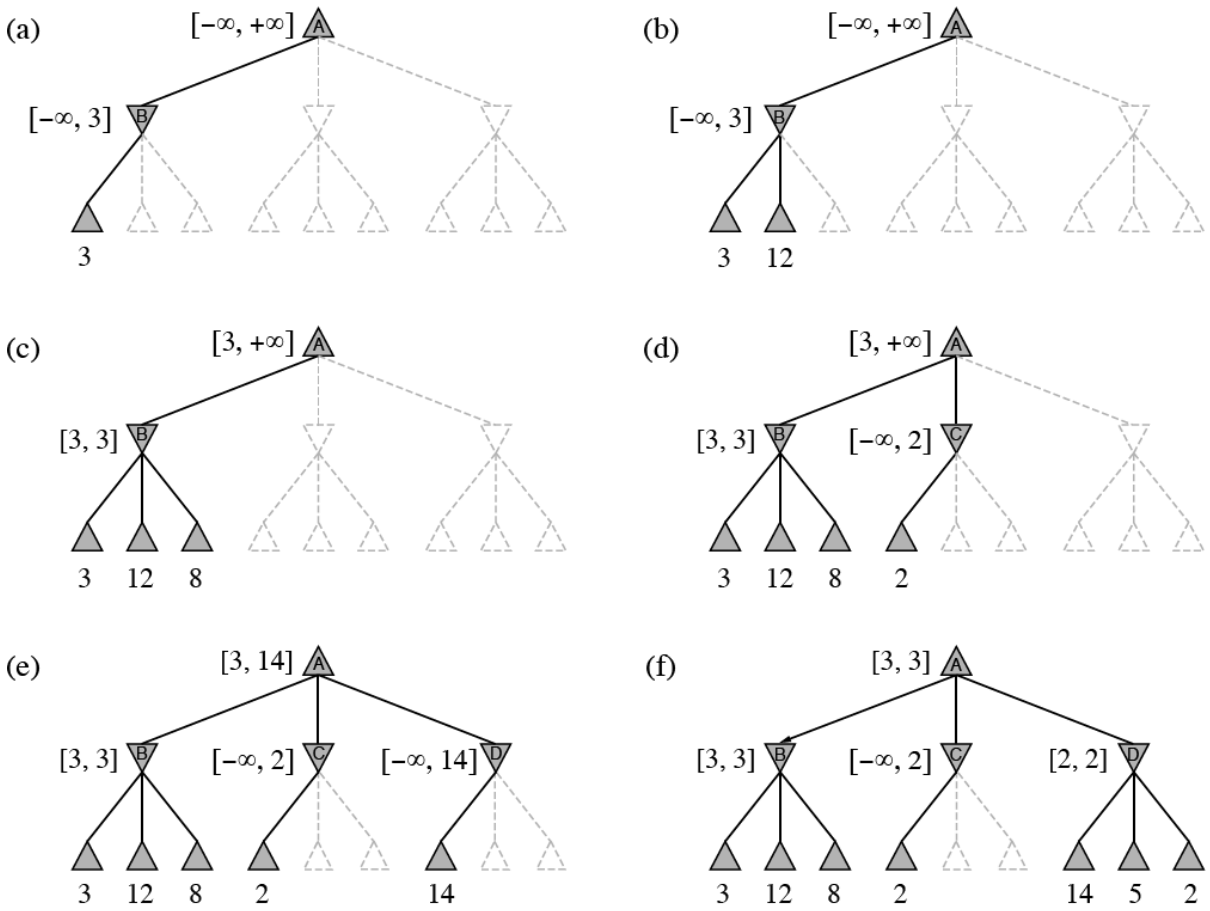
(e) The first leaf below D has the value 14, so D is worth at most 14. This is still higher than MAX's best alternative (I.e. 3), so we need to keep exploring D's successors. Notice also that we now have bounds on all the successors of the root, so the roots value is at most 14.

α-β pruning



(f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

α-β pruning



Note: we could not prune any successors of D at all because the worst successors (from the point of view of min) were generated first. If the third successor had been generated first, we would have been able to prune the other two. This highlights that α-β pruning is dependent on the order in which successors are examined.

α-β pruning

The outcome of the α-β pruning example we just stepped through was that we were able to identify the minimax decision without evaluating two of the leaf nodes.

α-β pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves.

Transposition Tables

Another way to prune search is to recognise repeated states.

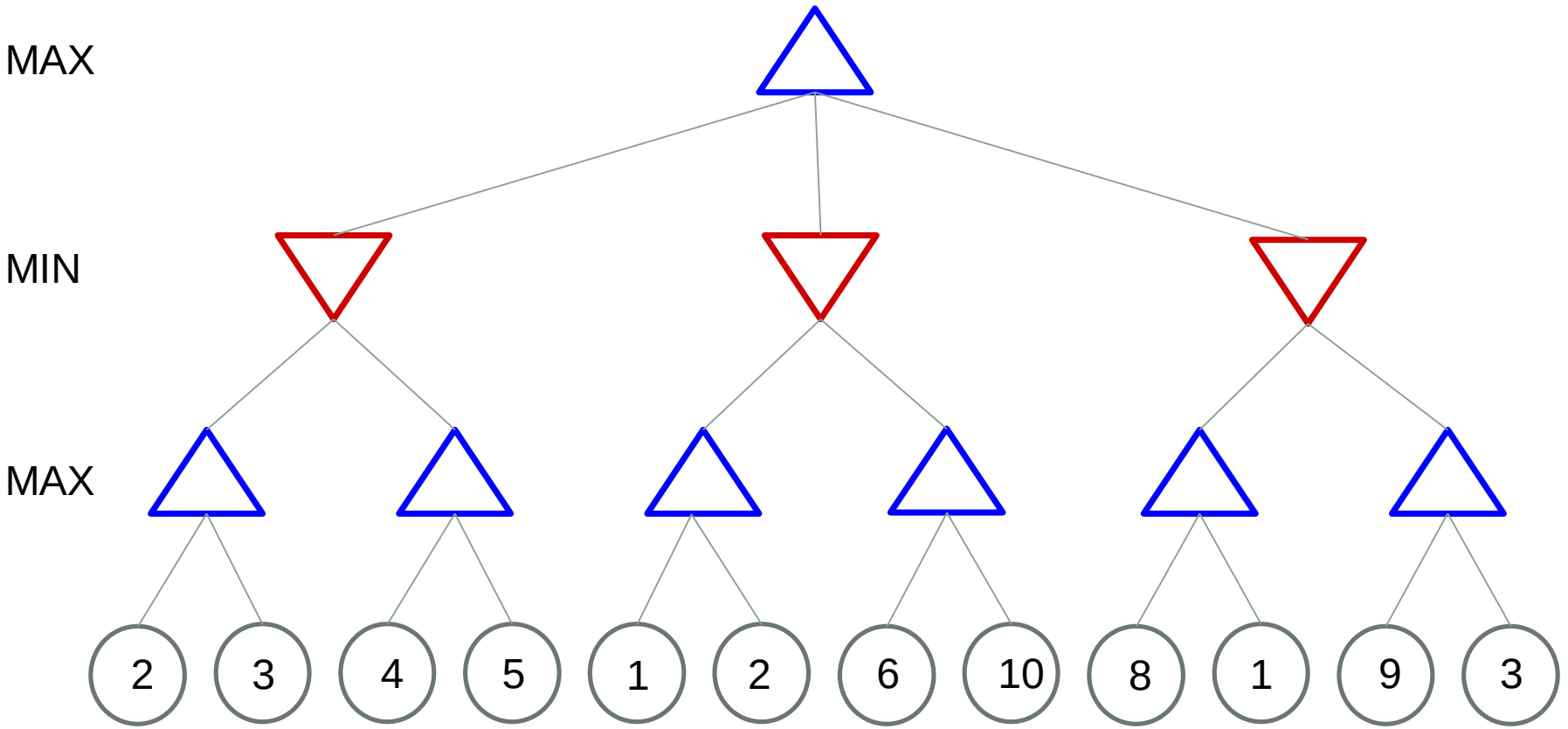
In games repeated states occur frequently, because of transpositions - different permutations of the move sequence that end up in the same position.

It is worthwhile to store the evaluation of each new position in a hash table so that we don't have to re-compute it on subsequent occurrences.

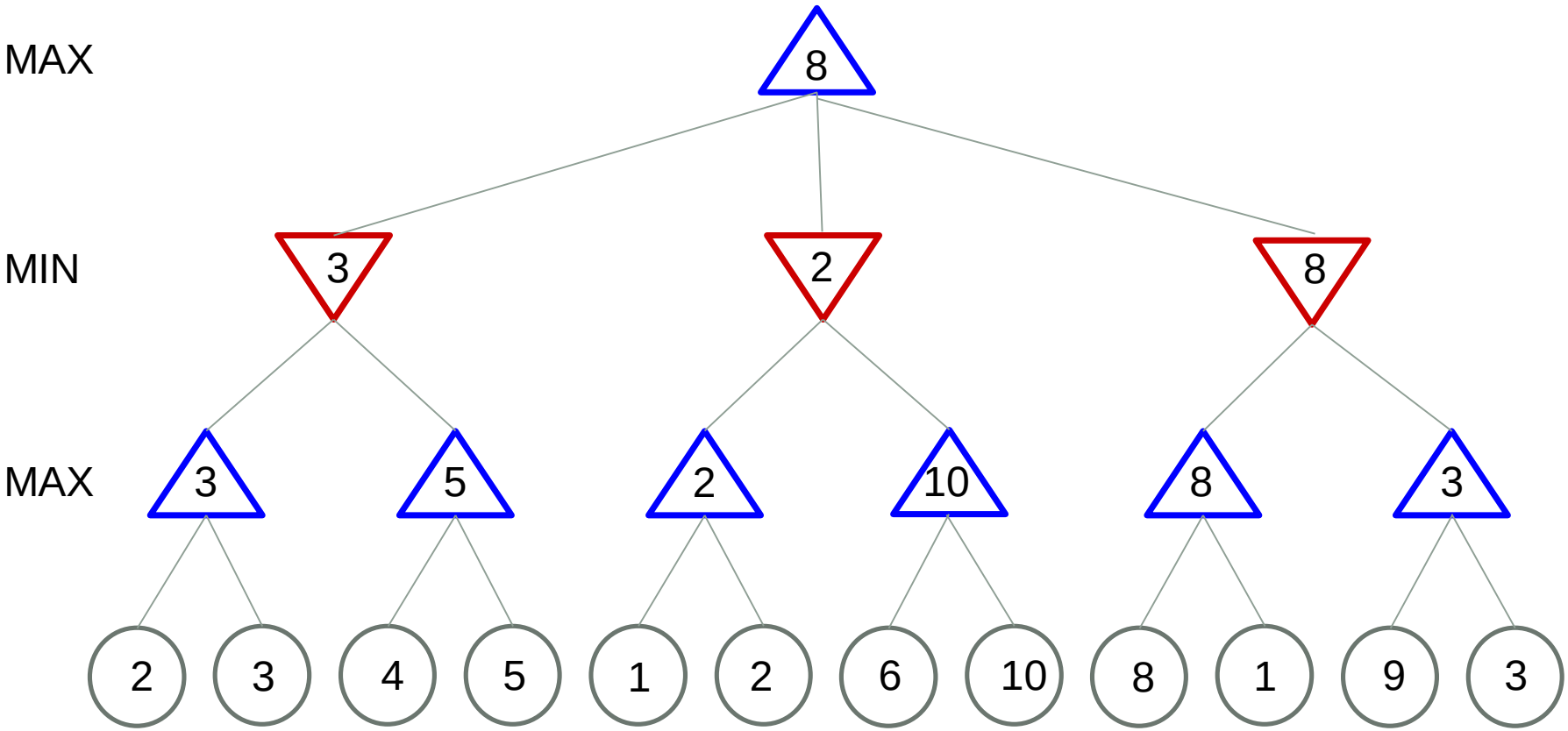
The table of previously seen positions has traditionally been called a transposition table.

EXERCISE

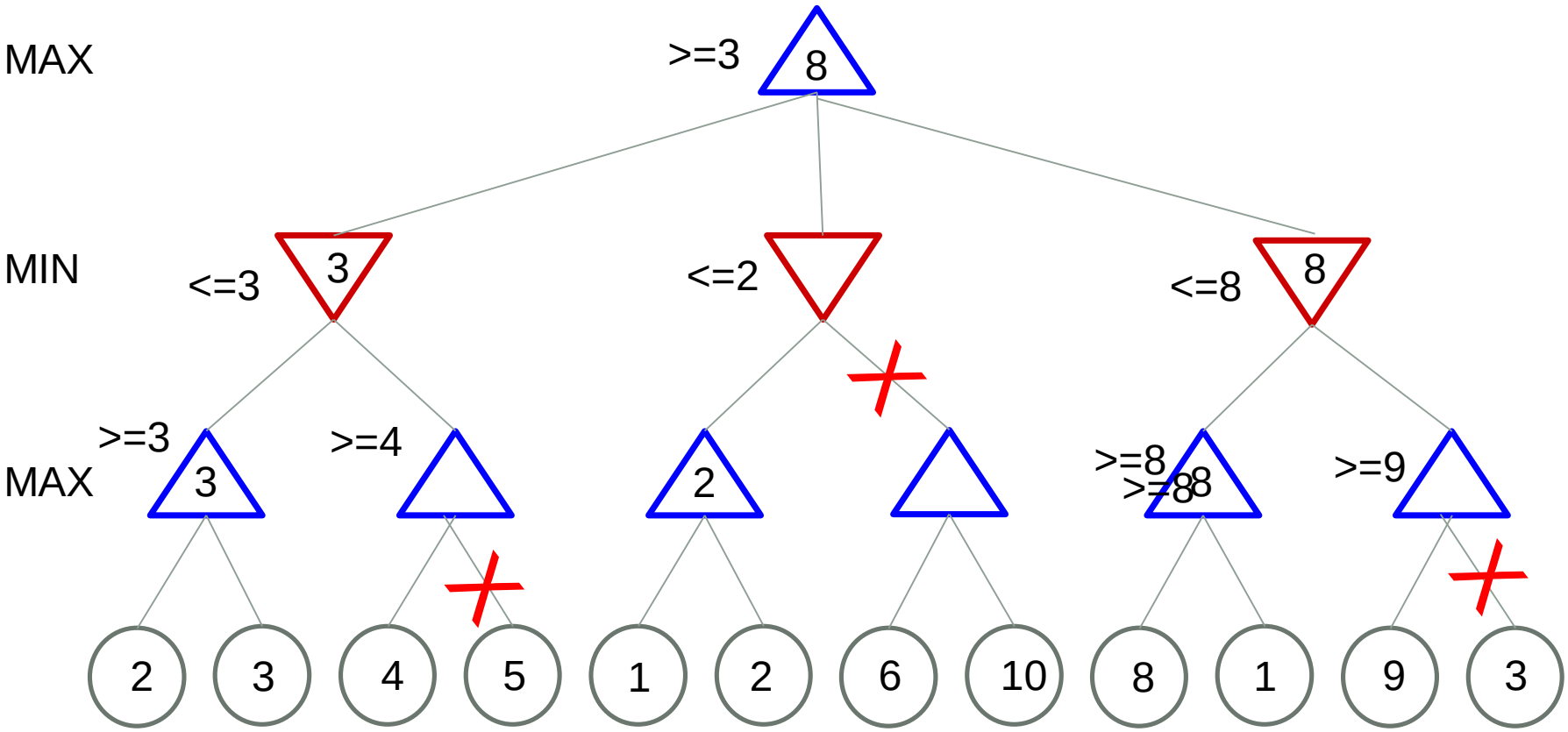
(1) Compute the minimax values for each node in the tree
(2) Perform a left-to-right alpha-beta prune



EXERCISE (1) – SOLN.



EXERCISE (2) – SOLN.



Summary

Games

Optimal Decisions: Minimax algorithm

α-β Pruning